B+-Tree

Donghui Zhang¹, Kenneth Paul Baclawski¹, Vassilis J. Tsotras²
¹Northeastern University, Boston, MA, USA
²University of California-Riverside, Riverside, CA, USA

Synonyms
B-tree

Definition
The B+-tree is a disk-based, paginated, dynamically updateable, balanced, and tree-like index structure. It supports the exact match query as well as insertion/deletion operations in $O(\log_p n)$ I/Os, where $n$ is the number of records in the tree and $p$ is the page capacity in number of records. It also supports the range searches in $O(\log_p n + t/p)$ I/Os, where $t$ is the number of records in the query result.

Historical Background
The binary search tree is a well-known data structure. When the data volume is so large that the tree does not fit in main memory, a disk-based search tree is necessary. The most commonly used disk-based search trees are the B-tree and its variations. Originally invented by Bayer and McCreight [2], the B-tree may be regarded as an extension of the balanced binary tree, since a B-tree is always balanced (i.e., all leaf nodes are on the same level). Since each disk access retrieves or updates an entire block of information between memory and disk rather than a few bytes, a node of the B-tree is expanded to hold more than two child pointers, up to the block capacity. To guarantee worst-case performance, the B-tree requires that every node (except the root) has to be at least half full. Because of this requirement, an exact match query, insertion or deletion operation must access at most $O(\log_p n)$ nodes, where $p$ is the page capacity in number of child pointers, and $n$ is the number of objects. The most popular variation of the B-tree is the B+-tree [3,4]. In a B+-tree, objects are stored only at the leaf level, and the leaf nodes are organized into a double linked list. As such, the B+-tree can be seen as an extension of the Indexed Sequential Access Method (ISAM), a static (and thus possibly unbalanced if updates take place) disk-based search tree proposed by IBM in the mid 1960’s.

Foundations

Structure
The B+-tree is a tree structure where every node corresponds to a disk block and which satisfies the following properties:

- The tree is balanced, i.e., every leaf node has the same depth.
- An internal node stores a list of keys and a list of pointers. The number of pointers is one more than the number of keys. Every node corresponds to a key range. The key range of an internal node with $k$ keys is partitioned into $k+1$ sub-ranges, one for each child node. For instance, suppose that the root node has exactly two keys, 100 and 200. The key range of the root node is divided into three sub-ranges $(−∞, 100), (100, 200)$ and $(200, +∞)$. Note that a key in an internal node does not need to occur as the key of any leaf record. Such a key serves only as a means of defining a sub-range.
- A leaf node stores a list of records, each having a key and some value.
- Every node except the root node is at least half full. For example suppose that an internal node can hold up to $p$ child pointers (and $p$-1 keys, of course) and a leaf node can hold up to $r$ records. The half full requirement says any internal node (except the root) must contain at least $[p/2]$ child pointers and any leaf node (except the root) must contain at least $[r/2]$ records.
- If the root node is an internal node, it must have at least two child pointers.
- All the leaf nodes are organized, in increasing key order, into a double linked list.

An example B+-tree is given in Fig. 1. It is assumed that every node has between two and four entries. In a leaf node, an entry is simply a record. In an internal node, an entry is a pair of (key, child pointer), where the key for the first entry is NULL. To differentiate a leaf entry (which corresponds to an actual record) from a key in an index entry, each leaf entry is followed by a "*".

Query Processing
The B+-tree efficiently supports not only exact-match queries, which find the record with a given key, but also range queries, which find the records whose keys are in a given range. To perform an exact-match query, the B+-tree follows a single path from the root to a leaf. In the root node, there is a single child pointer whose key range contains the specified key. If one follows the child pointer to the corresponding child node, inside the child node there is also a single child pointer whose key range contains the desired key. Eventually, one reaches a leaf node. The desired record, if it exists, must be located in this node. As an example, Fig. 1 shows the search path if one searches for the record with key = 41. Besides exact-match queries, the B+-tree also supports range queries. That is, one can efficiently find all records whose keys belong to a range \( R \). In order to do so, all the leaf nodes of a B+-tree are linked together. To search for all records whose keys are in the range \( R = [\text{low}, \text{high}] \), one performs an exact match query for key = \text{low}. This leads to a leaf node. One examines all records in this leaf node, and then follows the sibling link to the next leaf node, and so on. The algorithm stops when a record with key > \text{high} is encountered. An example is shown in Fig. 2.

Insertion
To insert a new record, the B+-tree first performs an exact-match query to locate the leaf node where the record should be stored, then the record is stored in the leaf node if there is enough space available. If there is not enough space, the leaf node is split. A new node is allocated, and half of the records, the ones with the larger keys in the overflowing node, are moved to the new node. A new index entry (the smallest key in the new node and a pointer to the new node) is inserted into the parent node. This may, in turn, cause

---

B+-Tree. Figure 1. Illustration of the B+-tree and exact-match query processing. To search for a record with key = 41, nodes \( I_1 \), \( I_2 \) and \( B \) are examined.

B+-Tree. Figure 2. Illustration of the range query algorithm in the B+-tree. To search for all records with keys in the range \([41,60]\), the first step is to find the leaf node containing \( 41^* \) (\( I_1 \), \( I_2 \) and \( B \) are examined). The second step is to follow the right-sibling pointers between leaf nodes and examine nodes \( C \) and \( D \). The algorithm stops at \( D \) because a record with key > 60 is found.
the parent node to overflow, and so on. In the worst case, all nodes along the insertion path are split. If the root node is split into two, a new root node is allocated and therefore the height of the tree increases by one.

As an example, Fig. 3 shows an intermediate result of inserting record 92* into Fig. 1. In particular, the example illustrates that splitting a leaf node results in a “copy up” operation. The result is intermediate because the parent node $I_3$ will also be split.

The complete result after inserting 92* is shown in Fig. 4. Here the overflowing internal node $I_3$ is split. In particular, the example illustrates that splitting an internal node can result in a “push up” operation.

**Deletion**

To delete a record from the B+-tree, one first uses the exact-match query algorithm to locate the leaf node that contains the record, and then the record is removed from the leaf node. If the node is at least half full, the algorithm finishes. Otherwise, the algorithm tries to re-distribute records between an immediate sibling node and the underflowing node. If redistribution is not possible, the underflowing node is merged with an immediate sibling node. Note that this merge is always possible.

As an example, Fig. 5 shows the intermediate result of deleting record 41* from the B+-tree shown in Fig. 4. Note that when merging two leaf nodes, a key in the parent node is discarded, which is the reverse operation of copy up. The result is intermediate because node $I_2$ is also underflowing.

Figure 6 illustrates the final result of deleting 41*. The underflow of node $I_2$ is handled by merging $I_2$ with $I_3$. This merge causes key 51 from the parent node to be dragged down to $I_2$. It is the reverse operation of push up.

---

**B+-Tree. Figure 3.** Intermediate result of inserting record 92* into Fig. 1. The leaf node $F$ is split. The smallest key in the new node $G$, which is 90, is copied up to the parent node.

**B+-Tree. Figure 4.** Continued from Fig. 3. Final result of inserting record 92*. The overflowing internal node $I_3$ is split. The middle key, 72, is pushed up to the parent node.

**B+-Tree. Figure 5.** Intermediate result of deleting 41* from Fig. 4. Node $B$ is merged with node $A$. Key 40 (as well as the pointer to node $B$) is discarded from the parent node.
Comparison with Some Other Index Structures
Compared with the B-tree, the B+-tree stores all records at the leaf level, and organizes the leaf nodes into a double linked list. This enables efficient range queries. Compared with ISAM, the B+-tree is a fully dynamic structure that balances itself nicely as records are inserted or deleted, without the need for overflow pages. Compared with external hashing schemes such as Linear Hashing and Extendible Hashing, the B+-tree can guarantee logarithmic query cost in the worst case (while hashing schemes have linear worst-case cost, although this is very unlikely), and can efficiently support range queries (whereas hashing schemes do not support range queries).

Key Applications
The B+-tree index has been implemented in most, if not all, relational database management systems such as Oracle, Microsoft SQL Server, IBM DB2, Informix, Sybase, and MySQL. Further, it is implemented in many filesystems including the NTFS filesystem (for Microsoft Windows), ReiserFS filesystem (for Unix and Linux), XFS filesystem (for IRIX and Linux), and the JFS2 filesystem (for AIX, OS/2 and Linux).

Future Directions
The impact of the B+-tree index is very significant. Many disk-based index structures, such as the R-tree [4] and k-d-B-tree [5] or their variants, are extensions of the B+-tree. Concurrency in B+-trees was studied in [6]. The Universal B-tree, which extends the B+-tree to index multi-dimensional objects, was studied in [1].

Cross-references
- Linear Hashing
- Rtree
- Tree-based Indexing

Recommended Reading

Backup
- Logging and Recovery

Backup and Restore
Kenichi Wada
Hitachi Limited, Tokyo, Japan

Synonyms
Backup copy

Definition
Backup is the action of collecting data stored on non-volatile storage media to aid recovery in case the original